Your Right t Know

Finding Leaks and Bottlenecks with a Wind ws NT PerfM n COM Object

Rick Anderson
Microsoft Corporation

January 1999

Summary: Discusses how to use Microsoft® Windows NT® performance monitoring to improve your program's performance. (16 printed pages)

Introduction

One of the great underused features of Microsoft Windows NT (soon to be Windows® 2000) is that it provides a rich API for creating and accessing various counters associated with system events and performance data. There are literally scores of these counters—they range from the number of remote users currently hooked up via Remote Access Service (RAS) to the maximum amount of virtual memory a process has used. Because you can get data on the amount of memory used and number of handles used, Windows NT performance monitoring is very useful for finding resource leaks in your programs.

Like all interesting APIs, however, these are somewhat difficult to use. In this article I'll show you how to use a class I wrote to easily access any performance counter from C++. (You can download the class from the Knowledge Base article "Sample: Using the PHD Class to Isolate Memory Leaks" (Q194655).)

But wait—wouldn't it be useful to access performance data from Microsoft Visual Basic®, Microsoft Visual J++™, and scripting languages? It would—so I wrapped the C++ class in an Active Template Library (ATL) Component Object Model (COM) component you could use from any language that supports COM components. I won't go into the details of how the class works, but I will describe its workings and show you how to wrap it as an ATL COM object—and even how to use it from Visual Basic.

Why Windows NT Performance Monitoring Is Cool

I've been involved in computer performance evaluation for most of my career. I worked on the original team developing test suites that eventually became the SPEC benchmarks (see http://www.spec.org/). Much of this work involved measuring and analyzing the performance of large numerical simulation programs on specific hardware. An application can be tuned to a specific computer by adjusting parameters that impact memory, disk I/O, and cache hits. To get performance information on an application, I would have to modify the source code with system calls to get memory and timing statistics. (There is always a trade-off to be made between memory usage, disk I/O, and algorithm complexity.)

Once an application was algorithmically tuned to a specific hardware platform, a new computer would come out with completely different memory capacity, I/O bandwidth, CPU speed, and many other factors that impacted program efficiency. If I only had a tool that could monitor system resources while the program ran, I could avoid the complexity of instrumenting the source code to log performance data. If only I'd been lucky enough to use Windows NT in those

days, I could have just used the Windows NT performance monitoring APIs. Now that I'm a support engineer for Microsoft, Windows NT performance monitoring plays a central role in my application analysis—especially in proving (or disproving) claims of memory leaks and other resource leaks.

About PerfMon

The Windows NT administrative tool PerfMon (from the Start icon, \Programs\Administrative Tools\Performance Monitor) is indispensable for monitoring system resources, application bottlenecks, and program efficiency. Common uses for PerfMon are to monitor how much memory an application is using, how badly a computer is paging, and how much CPU time a process is taking.

PerfMon can be used to log data, send alert messages to the Windows NT event log when a counter exceeds a preset bound, and even run a program when a counter goes over a limit you have set. This article will not discuss the PerfMon event functionality but will focus on exposing performance data.

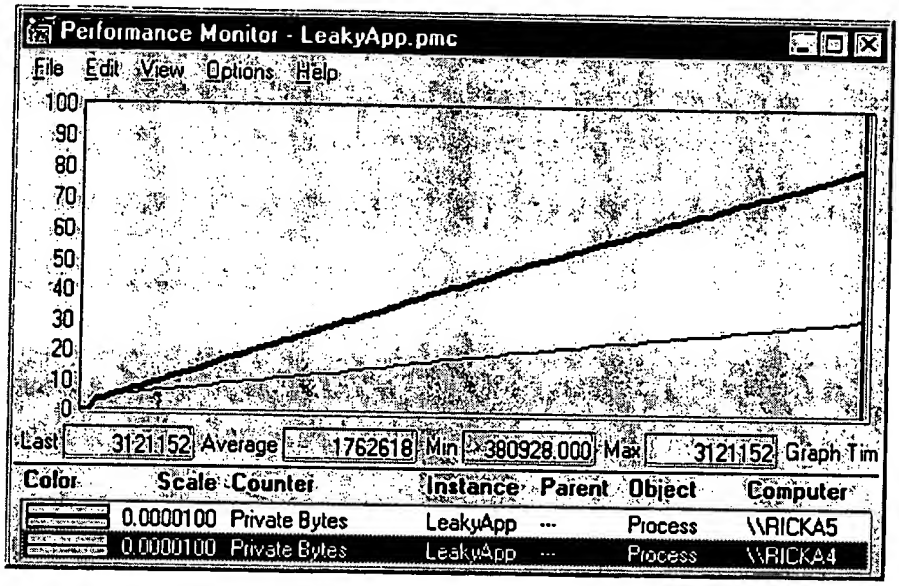

Figure 1. Performance Monitor displaying memory used by a process on two different machines

In Figure 1, PerfMon is displaying the "Private Bytes" my program (LeakyApp.exe) is using. Private Bytes is the current number of bytes a process has allocated that cannot be shared with other processes—in other words, how much nonshared memory your process is using. Private Bytes is the primary counter for measuring memory leaks and memory usage. Notice I'm running a copy of LeakyApp on two computers and displaying the memory usage for both applications in this PerfMon window. (As you can see, PerfMon is not limited to your local computer.)

Note that the longer these processes run, the more memory they use. Such a pattern of memory usage is typical of a process that is leaking memory. (Because applications don't always return memory immediately, such a graph is not *always* proof of a memory leak. But if memory usage increases without *ever* going constant, you have a definite memory leak.)

Analyzing a Leaky App

LeakyApp is a simple Microsoft Foundation Classes (MFC) database application that continuously throws MFC **CDBException** exceptions. It was a test application used to confirm that **CDBExceptions** were being handled correctly. A customer claimed there was a memory leak in Visual C++ version 5.0 exception handling and she wanted me to run her app on a Visual C++ 6.0 machine to see if this alleged bug had been fixed. The computer RICKA4 has Visual C++ 6.0 on it and RICKA5 has Visual C++ 5.0. From the trend just described, it indeed appears that Visual C++ 6.0 is leaking more slowly. The only problem is, RICKA4 is a 100-megahertz (MHz) Pentium with 128 megabytes (MB) of RAM. RICKA5 is a 450-MHz Pentium II with 256 MB of RAM. It may be that the program is leaking faster on RICKA5 because the CPU/memory combination allows the program to run much faster.

After perusing her test application, I discovered it was just a series of MFC **CDatabase** method calls intentionally called incorrectly so a **CDBException** would be thrown. The following code shows a typical sample:

```
try {        cdb.Open("Bogus String");     }
    catch( CDBException* e ){
        LogException(e->m_strError);
    }
```

The memory leak resulted from failure to delete the **CDBException** after it was caught. Deleting the exception after logging it totally plugged the leak—when I ran the fixed application, the memory usage graphs rose to a plateau, and then were flat. The correct code is as follows:

```
try {        cdb.Open("Bogus String");     }
    catch( CDBException* e ){
        LogException(e->m_strError);
        e->Delete();          // must free caught CException Objects
    }
```

Monitoring Resource Usage with PerfMon

PerfMon is the tool of choice to monitor application resource usage over time. In the preceding example it clearly showed a memory leak.

PerfMon displays a time versus value graph of counters you have selected. A counter is simply a number, such as the amount of memory a process uses, the percentage of CPU time used by a process, or the number of users connected via RAS.

Names of counters look something like network file paths. The counter name is like the file name in that it's the final element in the performance hierarchy. For example, `\\RICKA4\RickaD\work\ReadMe.htm` specifies `RICKA4` as the server, `RickaD` as the share name, `work` as a directory, and `ReadMe.htm` as a file. The performance hierarchy is generally `\\Computer\Object(Instance)\Counter`, so `\\ricka5\Process(LeakyApp)\Private Bytes` has `ricka5` as the computer, `Process` as the object (there is one process object for each running application on the computer), with `LeakyApp` being the process we are interested in. Finally, the counter we want to monitor is `Private Bytes`. So the counter named by this string is the Private Bytes counter of the LeakyApp process that is running on the `ricka5` computer.

You can read all about the PerfMon tool from the PerfMon Help menu. More in-depth information is available in the *Microsoft Windows NT 4.0 Resource Guide* Help. There are also several PerfMon articles available in the MSDN Library Online and CD. (Search on "Performance Monitor.")

Performance Monitoring Objects

The most commonly used objects are memory, process, processor, thread, and cache. Each of these objects has many counters, although some objects don't have instances. Most of us have single processor boxes, but for multiprocessor systems there is one processor instance for each CPU. Because all processors share the same memory on a Windows NT-based system, there is only one memory instance.

You can get information on any counter by selecting the counter in the **Add to Chart** dialog box and then pressing the **Explain>>** button. You can bring up the **Add to Chart** dialog box from the **Edit** menu. Unfortunately, online Help is not available for objects—just for counters. You can browse this dialog box to find all of the objects available on your system and the instances and counters available for each object.

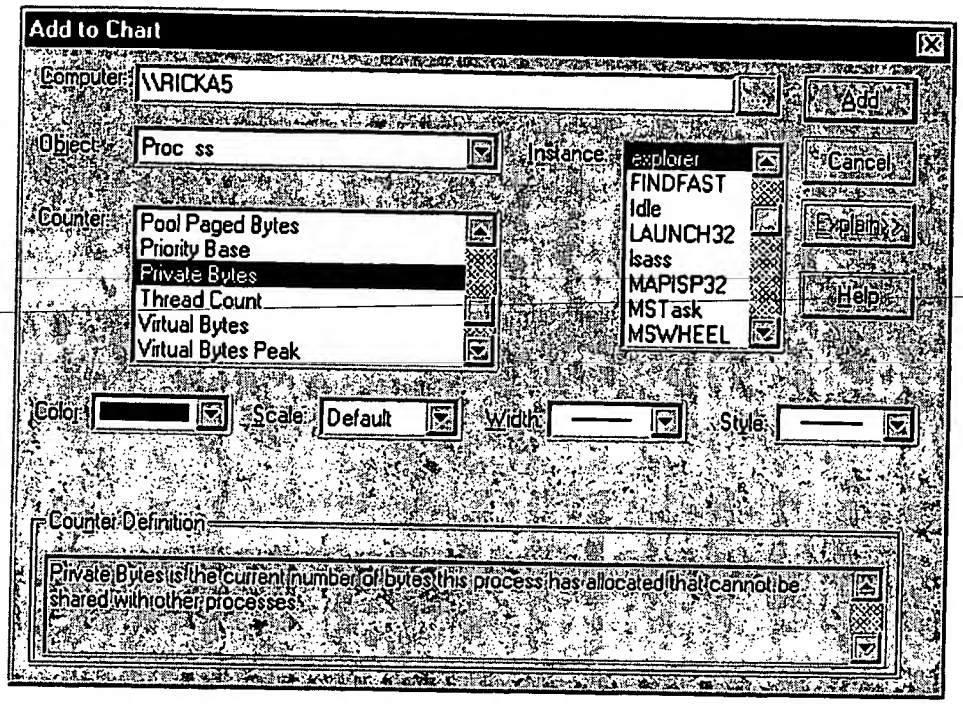

Figure 2. The Add to Chart dialog box

Important Counters

Windows NT Workstation comes with more than 15 objects. Windows NT Server has more. Applications like Microsoft SQL Server™ install additional counters. Each object has multiple instances and counters. So it's helpful to be able to focus on the useful counters.

The Memory object

The Memory object is almost self-explanatory and perhaps the most important resource to monitor. Because memory is more than a thousand times faster than disk, adding more memory to prevent swapping can be a cheap way to boost system performance. The memory counter "Pages/sec" is the number of pages read from the disk or written to the disk to resolve memory references to pages that were not in memory at the time of the reference. This is the primary counter to observe to determine if your system needs more physical memory to prevent paging for virtual memory.

The memory counter "Committed Bytes" displays the size of virtual memory (in bytes) that must have backup by disk or physical RAM. This counter can give you a good idea of how much physical RAM your system can use to prevent excessive paging.

The Process object

The Process object is probably the most useful to developers for monitoring applications. The Process object and a few others support the pseudo instance "_Total", which represents the total of all the instances. So `Process(_Total)\% Processor Time` is the counter indicating the percentage of processor time taken by all the processes on the system. The process counters I most frequently use are "Private Bytes," "Handle Count," "% Processor Time," and "Working Set."

I've already defined Private Bytes. Handle is the generic term for an opaque identifier to a resource. Common handles include file handles, window handles, and memory handles. Handles are an extremely precious resource, so leaking handles is more virulent than leaking memory. % Processor Time is the amount of CPU time a process is using. The Working Set is the amount of virtual memory the OS is maintaining for a process. Because Windows NT caches memory usage, freed memory for a process can stay in the process's working set. Many factors can influence the Working Set of a process, including memory requests of other processes, how the application was compiled (Visual C++ 6.0 supports the linker option /WS: AGGRESSIVE—this tells the system to aggressively trim the Working Set of the process when it is

not active), and memory usage patterns.

The User object

The User object represents a user of Windows NT. I use this object on a Windows NT Terminal Server shared by our group. When the Terminal Server seems sluggish, I can determine who is using all the processor time, memory (Private Bytes), or causing all the page faults (with the Page Faults/sec counter).

Specialized and Custom Counters

In addition to all the counters that come with Windows NT, you can add others to PerfMon, or even create your own application counters to monitor. I've already mentioned that SQL Server adds performance counters. Microsoft Transaction Server (MTS), Internet Information Server (IIS), and many other server applications add counters. In the August 1998 *Microsoft Systems Journal*, Jeffrey Richter presents a class for creating your own performance counters. Why would you want to add your own counters? Suppose you had an application that dynamically created many objects to satisfy a client app's requests. Many of the objects could be recycled, and thus reduce object creation overhead (perhaps at the cost of more memory or more disk). You could add counters for each object type in order to tune the caching algorithm based on server memory, load, disk speed, and so on.

Interpreting PerfMon Graphs: Another Example

In the graph to follow, the thick line shows paging (swapping) activity and the thin line shows CPU usage. PerfMon supports only one scale that all counters must use. In the preceding graph the scale is 0 to 200. Many counters, like % Processor Time, are a percentage. Percentage counter values are between 0 and 100. Because this particular graph has a maximum of 200, when the CPU is maxed out at 100 percent usage, the thin line is halfway on the scale—it can go no higher. PerfMon has no autoscaling feature, so you'll have to experiment to find a reasonable range. I'm most interested in paging, so for this particular hardware a scale of 200 works most of the time.

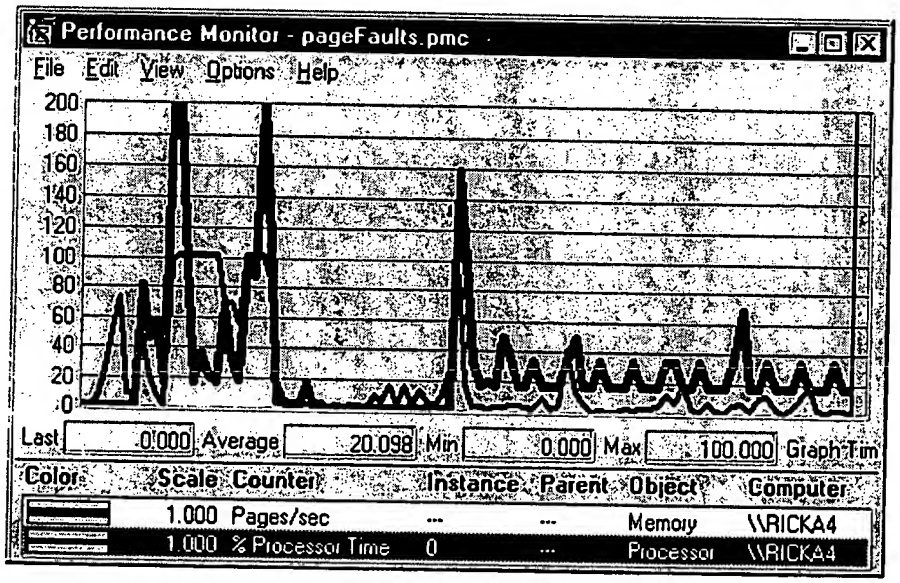

Figure 3. Example of a PerfMon graph

This PerfMon graph shows a lot of paging and CPU activity in the first one-fourth of the graph. At this time I started Microsoft Visual C++®, opened a project, and clicked the **Rebuild All** toolbar icon. The CPU went to full scale several times and paging was very high, even off the graph. In the middle of the graph there is no paging and the CPU stayed under 20 percent. I was running my application from the debugger at this time. The spike just past the middle was the result of rebuilding the application after a minor code change. This spike is much shorter than the previous build because Windows NT very efficiently caches files when you have surplus memory. (RAM plus Windows NT is your best friend.) If your computer shows high paging when you rebuild or switch to another task, take a PerfMon graph to your manager as proof you need more memory to finish that late assignment. The rest of the graph shows low paging and CPU activity as

I continued testing my application.

Using Performance Monitoring to Find Memory Leaks

I have been using PerfMon for some time to prove or disprove memory leaks in Microsoft APIs. With PerfMon, I could write a simple application that put the API in question in a loop, start up PerfMon, add the application's Private Bytes, and watch the trend. (Recall the steadily rising graph at the beginning of this article.) While this method is sometimes adequate, it has its limitations. PerfMon shows usage per time, not per call. More than 95 percent of the reported memory leaks turn out to be false (usually because the API was used incorrectly or a resource not explicitly freed), but in the 5 percent that are actual leaks, the developer would sometimes respond, "That's not a leak, that's just Windows NT caching data," or "my app is caching data."

Programmatic Access to Performance Data: The PHD Class

What I needed was a way to programmatically get PerfMon data. To solve this problem, I wrote a class wrapper around the Performance Data Helper Functions (the functions that PerfMon uses to get data) and turned the result into a Knowledge Base article. (See "SAMPLE: Using the PHD Class to Isolate Memory Leaks" (Q194655).)

The class has two constructors:

```
PHD(char **ppc, int nc,char *fileName=NULL,bool addDate=false,
    bool logState=false);
PHD(char *fileName=NULL,bool addDate=false, bool logState=false);
```

The parameters are:

- **ppc**

  The array of counters we want to monitor.

- **nc**

  The number of counters you are passing with ppc.

- **fileName**

  The name of the file to create.

- **addDate**

  A true value will append the current date to fileName.

- **logState**

  A true value will log the internal state of the PHD object. This is useful for debugging.

This is a very simple class, which needs only one method to write an entry to the log file:

```
void logData(int loopCnt);
```

The parameter is:

- **loopCnt**

  The count of the number of times you have called the API in question. The value you pass to *logData* is written to the Log file along with the current values of all the counters initialized in the PHD constructor.

With two constructors and only one method, the PHD class is very simple. (It's much more complicated underneath. We'll talk about how the PHD class uses the Windows NT Performance Data Functions in a later article—for now, let's just use the class.) You can view the PHD class as a black box, much as the PerfMon utility. You feed it exactly the same strings you feed the PerfMon utility, and it creates a Log file of the Counter data. Unlike PerfMon, which by default displays no counters, the default PHD constructor provides the following process counters for your application:

- Private Bytes

- Working Set
- Handle Count

Now any C++ programmer who suspects an API is leaking a resource can use PHD to get the definitive word. The following sample code shows a complete program that uses two instances of the PHD class, a default object and a specific object:

```
#include "rkLeak.h"

void main()
{
   char *myCntrs[]={
         "\\Memory\\committed bytes",
         "\\Processor(0)\\% Processor Time",
         "\\\\ricka5\\Processor(0)\\% Processor Time"
   };
   PHD phd2(myCntrs,sizeof(myCntrs)/sizeof(myCntrs[0]),"c:\\myMem");
   PHD phd1;

   const int arSize=4096;
   for (int i=0;i<5;i++){
      double *d = new double[arSize];
      d[0]= 1.;      d[arSize-1] = 2.;  // force to committed mem
      phd1.logData(i);
      phd2.logData(i);
   }
}
```

Warning The **PdhAddCounter()** function that the PHD constructor calls requires an exact counter string. For example, if you misspell a name, mess up the backslashes, or omit the space between the "%" and "Processor" (in the counter `"\\\\ricka5\\Processor(0)\\% Processor Time"`), the PHD constructor will stall for a few minutes and then assert. Use PerfMon to figure out the correct string to pass and set the **bLogState** argument of the PHD constructor to true. When **bLogState** is set to true the PHD class will log its internal state, including each counter string it successfully adds.

Notice in the code that I'm initializing data at the beginning and end of the allocated array. If you don't initialize this newly allocated memory, Windows NT will reserve the memory but not commit it. Because the memory is not committed, it won't show up in the Private Bytes or Committed Bytes counters. I don't need to initialize all the memory, just enough to ensure the newly allocated pages get written to.

I built the preceding code into an image called myApp.exe and ran it. Two output files were produced.

The default constructor instantiated as PHD1 created a file in my current working directory called myApp_Perf.log. The myApp_Perf.log has data for the default counters.

PHD2 created a tab-delimited file, "c:\myMem_myApp_Perf.log," with the following contents:

```
LoopCnt    committed bytes   % Processor Time   % Processor Time
0   170328064   82   21
1   170369024   0   0
2   170405888   16   16
3   170442752   5   3
4   170479616   0   0
```

You probably won't get the same numbers for % Processor Time. The Committed Bytes column shows a linear memory increase with each loop iteration. The raw numbers by themselves are boring, but because the file data is tab-delimited, you can paste it into Microsoft Excel, run the Chart Wizard, and a couple steps later you have a graph of the memory usage.

Making PHD int a COM Object

Even though C++ has the advantage when it comes to ease of creating resource leaks, Visual Basic and Java

applications are not immune from this problem. One Friday about 4:15 P.M. I got a hot tip on a component that was allegedly leaking memory from Visual Basic. Why not wrap PHD in a simple ATL control, and expose all this wonderful data to Visual Basic and Java? While I was at it, I would keep the ATL Object Wizard default of a dual interface, save myself one mouse click, and add Visual Basic Scripting Edition (VBScript) and all the other scripting clients to the club. The ATL Wizards are so complete, 15 minutes later the R1PHDMod component was functional and plugged into my simple Visual Basic app. It showed there was indeed a memory leak.

If you have Visual C++ 6.0 or Visual C++ 5.0, you can build the R1PHDMod component. If you don't have Visual C++ you can download it and use it directly. After unzipping R1PHDMod.dll to your favorite directory, register it (run `regsrv32 R1PHDMod`), and the component is armed and fully operational.

Wrapping a C++ Class in an ATL Control

The PHD class is a great candidate for wrapping in a COM object. It has a simple API and provides useful data to non-C++ programmers. Because the PHD class has no visual or graphics objects, we can put it in a simple ATL control. Before we build the COM object, we will define the parameters that need to be passed to the component. Because COM doesn't support passing parameters to an object's constructor, we have to add initialization functions. For the first constructor:

```
PHD(char **ppc, int nc,char *fileName=NULL,bool addDate=false,
    bool logState=false);
```

... we add the following initialization function (described here in IDL):

```
initLog([in] VARIANT *CntrArrayIn,[in] BSTR LogFileName, [in, optional]
      VARIANT  bDate, [in, optional] VARIANT  bState)
```

The second constructor is identical to the first, except that the first parameter is omitted.

For the logging method, the C++ class code:

```
void logData(int loopCnt);
```

... becomes, in IDL:

```
logCnt([in, optional, defaultvalue(-1)] long cnt)
```

The following steps outline the R1PHDMod creation. This is just a brief outline. If you need more explanation, see the ATL and Visual Studio documentation or "Dr. GUI and ATL."

1. Create a new ATL COM AppWizard project and name it "R1PHDmod," accepting all the defaults.
2. Insert a "New ATL Object." Select the default **Simple Object**. Use **rx4Leaks** for the Short Name. On the **Attributes** tab, show your support for robust error handling by checking the support **ISupportErrorInfo**.
3. In the Workspace window select the **ClassView** tab. Expand the "R1PHDmod Classes" so you can see the **Irx4Leaks** interface. We will be adding two **Init** methods corresponding to the two PHD constructors, and a **LogCnt** method for the **PHD::logData**.

   A C-language character array corresponds to a BSTR in COM. I'll arbitrarily choose to put the Boolean arguments in VARIANTs. All the arguments to the PHD methods are input, or IN, arguments; no data is returned in the arguments. Simple types, like short, allow the IDL keyword **defaultvalue**, so I could have described the **bAddDate** IDL as:

   ```
   [in, optional, defaultvalue(0)] short
   ```

   You can't use **defaultvalue** with a VARIANT, but you can use the keyword ***optional***. An optional argument not supplied by the client is sent with type VT_ERROR. The array of counters will be sent as a BSTR array with the BSTR array in a VARIANT. Except for the array of BSTRs, all of our arguments are very simple and straightforward.

4. Call the method corresponding to the PHD default constructor **InitEzL g** and use the IDL I just defined. (Right-click the **Irx4Leaks** Interface and select **Add Method ....**) Add the method **InitL g**, which corresponds to the first constructor using the IDL just defined. Finally, add a **L gCnt** method for the PHD **LogData** method.

   I know it's vogue in some circles to use cryptic names for arguments, but it's definitely in bad taste when it comes to COM. Modern tools like Visual Basic 6.0 have Microsoft IntelliSense®, so a developer using your component from Visual Basic 6.0 doesn't need to guess what your arguments represent. When she types "myR1PHD.initEzLog" the IDE will display:

```
initEzLog(LogFileName as String,[bAddDate],[bAddState])
```

5. Now all we have to do is hook up the plumbing. Add the following includes and pragmas to the rx4Leaks.cpp file:

```
#include "rkLeak.h"
#include <strstrea.h>
#pragma comment(lib, "pdh.lib")     // don't need to put pdh.lib in
                                    // project settings
#pragma warning( disable : 4800)  // Disable bool warning
```

   The rkLeak.h is the include for the PHD class. To successfully link to the pdh.dll, you need the "phd.lib" file. By specifying it in the #pragma, we don't have to manually set it in our project settings. Because R1PHDmod is a debugging component, we don't want to be warned about Boolean conversion slowing down the code.

A Macro for building and debugging components

Next I'll introduce a macro that is useful for building and debugging components. Add the following macro to the top of the rx4Leaks.cpp file:

```
#define ERR_MSG(x) { strstream str; str << x << "\n" << __FILE__ << \
  ":" << __LINE__; str.str()[str.pcount()]=0; \
  Error(_T(str.str()),IID_Irx4Leaks); return E_FAIL;}
```

This macro is useful because it allows us to easily create and return complete error information to the client. Because it uses srtstreams, you can do something like the following:

```
if( arg < 0 )
   ERR_MSG(" you passed " << arg << " should have used" << someValue );
```

If the client passes -1 for the argument, the error message will look like this:

```
Run Time Error (0x80004005)
you passed -1 you should have used 99
C:\YourPath\RX4Leaks.cpp:55
```

Using this error information, we can quickly go to line 55 of the rx4Leaks.cpp file to debug our code.

The next COM interface you write probably won't be called rx4Leaks, so you won't be able to cut and paste the ERR_MSG macro without a bit of editing. The IID_I* globally unique identifier (GUID) you'll need is in the **InterfaceSupportsErrorInfo** method, the first method in your interface implementation .cpp file.

Next, add the handy global utility function **BgetBool**:

```
bool bgetBool( VARIANT v ){
       if (v.vt == VT_EMPTY || v.vt == VT_ERROR )
           // if client did not send this
           return false;
  // our convention is false
       return v.boolVal;
}
```

We will arbitrarily decide that Boolean optional arguments that are not supplied by the client have a default value of

false.

Add code to the **InitEzLog** method so it looks like the following:

```
STDMETHODIMP Crx4Leaks::initEzLog(BSTR LogFilename, VARIANT bDate, VARIANT bState)
{
    USES_CONVERSION;            // required for W2A macro
    bool bD = bgetBool(bDate);
    bool bS = bgetBool(bDate);

    if(m_phd)
        ERR_MSG("initLog previously called");

    m_phd = new PHD(W2A(LogFilename),bD, bS );

    if(!m_phd)
        ERR_MSG("new PHD() Allocation Failure");
    return S_OK;
}
```

We use the macros USES_CONVERSION and W2A to convert the BSTR to an ASCII string that the PHD class takes.

Now add the following code to the **LogCnt** method so it looks as follows:

```
STDMETHODIMP Crx4Leaks::logCnt(long cnt)
{
    if(!m_phd){
        m_phd = new PHD();
        if(!m_phd)
            ERR_MSG("new PHD() Allocation Failure");
    }
    static int sCnt;
    int theCnt = (cnt== -1) ? sCnt++ : cnt;
    m_phd->logData(theCnt);
    return S_OK;
}
```

The **LogCnt()** method allows the lazy programmer to use the component without explicitly calling an initialization method. The **LogCnt** argument is optional, so if it's not supplied, we will keep track of the count ourselves.

Next, add a forward declaration to the PHD class to your ATL class definition (in the file rx4Leaks.h):

```
class PHD;
```

Add a pointer to the PHD class (m_phd) and initialize it in the constructor so your code looks like this:

```
public:
        PHD *m_phd;
        Crx4Leaks() : m_phd(0)
   {
   }
```

Build the component now. If you have a fast computer and you didn't make any syntax errors, you should have 20 seconds to spare in our race to get a working component in 15 minutes.

The Visual Basic Sample *Hello Memory Leak*

To test our new control, we'll write a small Visual Basic sample. First, create a new Visual Basic EXE Project. From the **VB Project** menu, select **References**, type "R" and you will go right to the R1PHDMod 1.0 type library. Check **R1PHDmod type library**. Add a command button to the form, and the following:

```
Option Explicit
Dim rx As R1PHDMODLib.rx4Leaks
Private GL As Variant
```

```
Private G_cnt As Long

Private Sub Command1_Click()
G_cnt = G_cnt + 1
ReDim GL(2048 * G_cnt)
rx.logCnt
End Sub

Private Sub Form_Load()
Set rx = New R1PHDMODLib.rx4Leaks
End Sub
```

You can run the program from Visual Basic, but to see a leak you must make an executable. (The default **Init** method adds counters to the process that loaded it. When you run from Visual Basic, you are monitoring the Visual Basic 6.0 process, not the application you just wrote.) Create the default **Project1.exe** and run it. Click the **Command1** button a few times. You will be rewarded with the file "Project1_Perf.log," which does indeed show a memory leak.

Enjoy using the PHD class and the R1PHDMod component to find resource leaks, or just to monitor the performance of your program. Next time we'll describe how the PHD class and the Windows NT performance monitoring APIs it calls work. We'll also complete the R1PHDMod component and demonstrate it with Java and VBScript clients.